React 中的高阶组件

高阶组件（HOC）是什么？

HOC 是 React 中的一种进阶用法，函数（组件）接收组件作为参数并返回一个新的组件。

```
function composeComponent(Component) {
    return class extends React.Component {
        render() {
            return <Component />
        }
    }

}
```

在这里，函数 composeComponent 接收了一个 Component 变量作为参数并返回一个 ES6 class 定义的组件。返回的 class 组件中使用了参数中的 Component 变量。Component 参数会是一个 React 组件，它将被返回的 class 组件调用。

例如：

```
class CatComponent extends React.Component {
    render() {
        return <div>Cat Component</div>
    }
}
```

我们有一个 CatComponent 组件，渲染结果如下：

```
Cat Component
```

我们可以将 CatComponet 作为参数传递给 composeComponent 函数得到另一个组件：

```
const composedCatComponent = composeComponent(CatComponent)
```

composedCatComponent 组件也能够被渲染：

```
<composedCatComponent />
```

渲染结果如下：

```
Cat Component
```

这和 JS 中的高阶函数是类似的。

高阶函数

高阶函数是 JS 中的一种模式，函数接收一个函数作为参数并返回另一个函数作为结果。因为 JS 本身的语法特性使得这是可行的。这意味着以下类型的数据：

- objects
- arrays
- strings
- numbers
- boolean
- functions

都可以作为参数传递给函数，也可以从函数中返回。

```
function mul(x) {
    return (y) => {
        return x * y
    }
}
const mulTwo = mul(2)

mulTwo(2) // 4
mulTwo(3) // 6
mulTwo(4) // 8
mulTwo(5) // 10
```

mul 函数返回了一个函数，该函数在闭包中捕获变量 x 的值。现在，返回的函数可以使用这个 x。mul 现在就是一个高阶函数，因为它返回了一个函数。这意味着我们可以调用它通过传递不同的参数来构造其它更具体的函数。

我们可以用它来创建一个函数，返回参数的 3 倍：

```
function mul(x) {
    return (y) => {
        return x * y
    }
}
const triple = mul(3)

triple(2) // 6
triple(3) // 9
triple(4) // 12
triple(5) // 15
```

那高阶函数和高阶组件有什么好处呢？当我们发现自己一遍又一遍地重复相同的逻辑时。我们需要找到一种方法把相同的逻辑封装在一起，然后从那里调用它。高阶函数就提供了一个我们可以用来实现它的模式。

从上面的例子中，如果在我们的程序中需要多次乘以 3，我们就可以创建一个函数，返回另一个乘以参数 3 倍的函数，所以每当我们需要进行 3 倍乘法时，我们可以简单地调用通过传递参数 3 获得的 triple 函数。

使用高阶组件（HOC）

所以，在 React 中使用高阶组件又有什么好处呢？

同样，我们在 React 项目的编程过程中，也可能会发现自己一次又一次地重复相同的逻辑。

例如，我们有一个应用程序是用来查看和编辑文档的。我们希望对应用程序的用户进行身份验证，这样只有经过身份验证的用户才能访问主页、编辑文档、查看文档或删除文档。我们的路由是这样设计的：

```
<Route path="/" component={App}>
    <Route path="/dashboard" component={Documents}/>
    <Route path="document/:id/view" component={ViewDocument} />
    <Route path="documents/:id/delete" component={DelDocument} />
    <Route path="documents/:id/edit" component={EditDocument}/>
</Route>
```

我们必须在 Documents 组件中进行验证，这样只有通过验证的用户才能访问它。如下：

```
class Doucments extends React.Component {
```

```
    componentwillMount() {
        if(!this.props.isAuth){
            this.context.router.push("/")
        }
    }
    componentWillUpdate(nextProps) {
        if(!nextProps.isAuth) {
            this.context.router.push("/")
        }
    }
    render() {
        return <div>Documents Paegs!!!</div>
    }
}

function mapstateToProps(state) {
    isAuth: state.auth
}
export default connect(mapStateToProps)(Documents)
```

state.auth 保存了用户的验证状态。如果用户没有通过验证，它的值是 false，如果通过验证，它将是 true。connect 函数将 state.auth 映射到组件 props 对象的 isAuth。然后，当组件将要挂载到 DOM 上时，componentWillMount 被触发，因此我们检查 props 的 isAuth 是否为真。如果为真，组件会继续渲染；否则，则该方法会将路由切换到 “/” 路径，从而使得我们的浏览器在渲染 Documents 组件时被重定向到了首页，进而有效地阻止了未通过验证的用户对它的访问。

当组件在初始渲染之后再次渲染时，我们只在 componentWillUpdate 中执行相同的操作，以检查用户是否仍然具有授权，如果没有，则同样重定向到首页。

然后，我们可以在 ViewDocument 组件中做同样的处理：

```
class ViewDoucment extends React.Component {
    componentwillMount() {
        if(!this.props.isAuth){
            this.context.router.push("/")
        }
    }
    componentWillUpdate(nextProps) {
        if(!nextProps.isAuth) {
            this.context.router.push("/")
        }
    }
```

```
    render() {
        return <div>View Document Page!!!</div>
    }
}

function mapstateToProps(state) {
    isAuth: state.auth
}
export default connect(mapStateToProps)(ViewDocument)
```

在 EditDocument 组件中：

```
class EditDocument extends React.Component {
    componentwillMount() {
        if(!this.props.isAuth){
            this.context.router.push("/")
        }
    }
    componentWillUpdate(nextProps) {
        if(!nextProps.isAuth) {
            this.context.router.push("/")
        }
    }
    render() {
        return <div>Edit Document Page!!!</div>
    }
}

function mapstateToProps(state) {
    isAuth: state.auth
}
export default connect(mapStateToProps)(EditDocument)
```

在 DelDocument 组件中：

```
class DelDocument extends React.Component {
    componentwillMount() {
        if(!this.props.isAuth){
            this.context.router.push("/")
        }
    }
    componentWillUpdate(nextProps) {
        if(!nextProps.isAuth) {
```

```
            this.context.router.push("/")
        }
    }
    render() {
        return <div>Delete Document Page!!!</div>
    }
}

function mapstateToProps(state) {
    isAuth: state.auth
}
export default connect(mapStateToProps)(DelDocument)
```

不同的页面具有不同的功能，但它们的大部分实现逻辑是相同的。

在每个组件中的操作：

- 通过 react-redux 连接到 store 的 state。
- 将 state.auth 映射到组件的 props.isAuth 属性。
- 在 componentWillMount 中检查用户是否授权。
- 在 componentWillUpdate 中检查用户是否授权。

假设我们的项目扩展了更多其他的组件，我们发现我们在每个组件中都实现了上述的操作。这肯定会很无聊的。

我们需要找到只在一个地方实现逻辑的方式。最好的办法就是使用高阶组件（HOC）。

为此，我们将所有的逻辑封装到一个函数中，该函数将返回一个组件：

```
function requireAuthentication(composedComponent) {
    class Authentication extends React.Component {
        componentwillMount() {
            if(!this.props.isAuth){
                this.context.router.push("/")
            }
        }
        componentWillUpdate(nextProps) {
            if(!nextProps.isAuth) {
                this.context.router.push("/")
            }
        }
        render() {
            <ComposedComponent />
```

```
        }
    }
    function mapstateToProps(state) {
        isAuth: state.auth
    }
    return connect(mapStateToProps)(Authentication)
}
```

可以看到，我们将所有相同的逻辑都封装到了 Authentication 组件中。requireAuthentication 函数将把 Authentication 组件连接到 store 并返回它。然后，Authentication 组件将渲染通过 composedCompoennt 参数传入的组件。

我们的路由现在这样的:

```
<Route path="/" component={App}>
    <Route path="/dashboard"
component={requireAuthentication(Documents)}/>
    <Route path="document/:id/view"
component={requireAuthentication(ViewDocument)} />
    <Route path="documents/:id/delete"
component={requireAuthentication(DelDocument)} />
    <Route path="documents/:id/edit"
component={requireAuthentication(EditDocument)}/>
</Route>
```

因此，无论我们的应用程序将来会有多少条路由，我们都不用考虑向组件添加身份验证的逻辑，我们只需调用 requireAuthentication 函数，并将组件作为参数传递给它。

使用高阶组件（HOC）会有很多的好处。当你发现你在重复相同的逻辑时，你需要把相同的逻辑封装到一起，并使用高阶组件（HOC）。